



CREATE WITH DATA

FUNDAMENTALS OF HTML, SVG, CSS & JAVASCRIPT FOR DATA VISUALISATION

Peter Cook

Fundamentals of HTML, SVG, CSS and JavaScript for Data Visualisation

Peter Cook

This book is for sale at <http://leanpub.com/html-svg-css-js-for-data-visualisation>

This version was published on 2022-10-17



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2022 Peter Cook

Also By **Peter Cook**

[D3 Start to Finish](#)

Contents

1. Introduction	1
1.1 Setting up	1
1.2 CodePen	1
1.3 Stay in touch	2
1.4 Translators	2
2. Web languages: HTML, SVG, CSS & JavaScript	3
3. HTML	4
3.1 Headings	4
3.2 Lists	5
3.3 Images	6
3.4 <code><div></code> element	7
3.5 <code><svg></code> element	8
3.6 Wrapping up	8
3.7 Exercises	8
4. SVG	10
4.1 Adding SVG to a web page	10
4.2 SVG elements	11
4.3 SVG Transforms	16
4.4 Exercises	20
5. CSS	22
5.1 The Structure of CSS	22
5.2 CSS Units	23
5.3 CSS Properties	24
5.4 More CSS selectors	29
5.5 CSS for HTML layout	36
6. JavaScript	45
6.1 JavaScript variables	45

CONTENTS

6.2	JavaScript data types	47
6.3	JavaScript arrays	51
6.4	JavaScript objects	54
6.5	JavaScript operators	56
6.6	JavaScript conditionals	59
6.7	JavaScript functions	61
6.8	JavaScript iteration	64
6.9	JavaScript functions (advanced)	69
7.	Tools and Set-up	73
7.1	Web development tools	73
7.2	Example set-up	77

1. Introduction

Welcome! This book covers the fundamentals of HTML, SVG, CSS and JavaScript for those who are interested in visualising data. If you work with data you might spend most of your time using Excel, R or Python. You might be tasked with creating charts in a Python Jupyter notebook or creating a dashboard using R's Shiny or Python's Dash. Often these charts are built with HTML, SVG, CSS and JavaScript and a good understanding of these languages is useful when you need to dive deeper.

Or you might be interested in learning D3 or other JavaScript charting libraries. In which case this book can help you get up to speed with HTML, SVG, CSS and JavaScript.

This book doesn't assume prior knowledge of these languages, but it'll be helpful if you've some coding experience. Being familiar with a text editor (or IDE) will also be helpful.

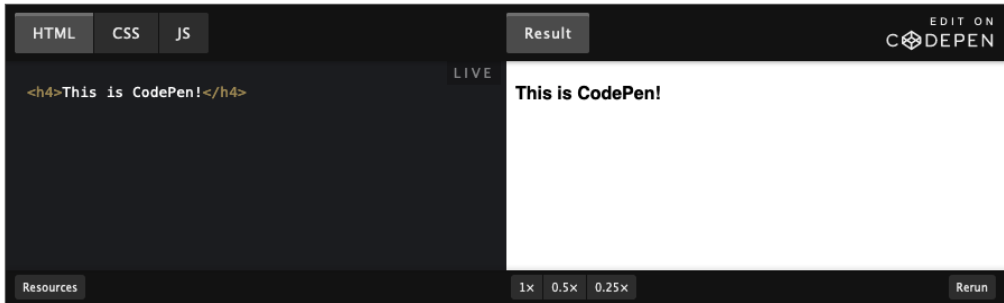
It's by no means a comprehensive tutorial on HTML, SVG, CSS and JavaScript. You'd need a much bigger book for that. Think of this book as presenting the minimum of what you ought to know if you're wanting to work with web based data visualisations. If you'd like to understand my other data visualisation books (such as D3 Start to Finish) you definitely need to understand the content in this book.

1.1 Setting up

In order to get started quickly I recommend using [CodePen](#)¹ to look at and experiment with code examples (see next section). At the end of this book, in the 'Tools and Set-up' chapter I show you how to set up tools so that you can write code locally (where the files are stored on your computer instead of on CodePen). This takes a bit of time to set up but I recommend this approach if you become more serious about web development.

1.2 CodePen

[CodePen](#)² is a web site that lets you experiment with HTML, SVG, CSS and JavaScript without having to set up any tools.



This is CodePen. You can visit this example at <https://codepen.io/createwithdata/pen/VwvpzOg>

At key points in the book there'll be a link to a CodePen 'pen' that contains a code example. If you visit the link you'll be able to view and experiment with the code. I encourage you to play around with the CodePen examples - there's very little you can break and learning through experimentation is very effective.

You can freely edit the CodePen examples from this book and your changes don't affect the original example. If you want to keep your changes, click Fork (at the bottom of the page) and create a CodePen account (or log in). Save your work by clicking the Save button at the top of the page.

1.3 Stay in touch

I love to stay in touch with my readers. One of the best ways to do this is via my mailing list. I send occasional messages containing useful information related to implementing data visualisations (e.g. using JavaScript or other tools). There'll also be discount codes for my other books. You can sign up [here](#)³.

1.4 Translators

If you're interested in translating Fundamentals of HTML, SVG, CSS and JavaScript for Data Visualisation to another language contact me at info@createwithdata.com and I'll be happy to discuss further.

Notes

¹ <https://codepen.io/>

² <https://codepen.io/>

³ <https://mailchi.mp/085e79e8fd7c/create-with-data-newsletter>

2. Web languages: HTML, SVG, CSS & JavaScript

Most websites and web applications (including data visualisations) are built from 4 languages: **HTML**, **SVG**, **CSS** and **JavaScript**.

HTML stands for Hyper Text Markup Language and describes the **content** of a webpage.

SVG stands for Scalable Vector Graphics and describes **shapes** such as circles, lines and rectangles. It's often used when visualising data (to describe bars, lines, circles, axes, legends etc.).

CSS stands for Cascading Style Sheets and describes the **style** (such as font weight, colour or size) and **position** of HTML and SVG elements.

JavaScript is a programming language that can **manipulate** HTML and SVG, do general **computation** (such as data manipulation) and handle **interaction**.

We'll look at each of these in the upcoming chapters.

3. HTML

HTML is a language that defines the **content** of a webpage. It defines a webpage using elements to define headings, paragraphs, images, links etc. For example a paragraph element looks like:

```
<p>This is a paragraph of text.</p>
```

The paragraph element consists of an **opening tag** `<p>`, some **content** and a **closing tag** `</p>`. Tags usually come in pairs. An opening tag looks like `<p>` and a closing tag looks like `</p>`.

When a web browser loads an HTML document it draws each element according to the type of element (paragraph, heading, image etc.). For example, the paragraph element above looks like the following when rendered in a web browser:

This is a paragraph of text.

A paragraph `<p>` element as rendered in a web browser

There's a large number of HTML elements but you can achieve a lot with just a few of them. This chapter looks at five common elements: headings, lists, images, `div` elements and `svg` elements. (These five elements are usually sufficient when visualising data.)

3.1 Headings

Heading elements are used for titles. There are six levels ranging from large (`<h1>`) to small (`<h6>`).

```
<h1>Main heading</h1>
<h2>Sub heading</h2>
<h3>Level 3 heading</h3>
<h4>Level 4 heading</h4>
<h5>Level 5 heading</h5>
<h6>Level 6 heading</h6>
```

These elements look like this when rendered in a web browser:

Main heading

Sub heading

Level 3 heading

Level 4 heading

Level 5 heading

Level 6 heading

Heading elements `<h1>` to `<h6>` rendered in a web browser



The web browser determines how each of these elements is styled. For example the `<h1>` is the largest and all the headings are rendered with bold text. You can set the style yourself using CSS (see the CSS chapter).

3.2 Lists

The list element `<ul>` is used to represent lists of items. Each item is represented by a list item element `<li>` and nested within the `<ul>` element:

```
<ul>
  <li>List item one</li>
  <li>List item two</li>
</ul>
```

Notice that we've indented the `<li>` elements. This is optional but is the usual way to indicate nested elements. When displayed in a browser the list looks like:

- List item one
- List item two

List elements `<ul>` and `<li>` rendered in a web browser



By default list items are bulleted and displayed as a vertical list. However you can change how a list looks using CSS. Often navigation menus are represented using a list and CSS used to hide the bullet points and position the items as desired.

3.3 Images

Images can be added to an HTML document using the `<img>` element. `<img>` elements require an **attribute** named `src` that specifies the URL of the image.



Attributes can be added to the opening tag of any HTML element and they specify additional information about the element. They take the form `name="value"` where the value is always in quotes.

Image elements don't have a closing tag. Here's an image element that displays an image located at <https://picsum.photos/id/127/300/200.jpg>:

```

```

This example looks like:



Image element `<img>` rendered in a web browser

Typically your image files will live alongside your source code. For example, you might have an image file `my-image.jpg` within a directory named `images`. In this case you'd write:

```

```

By default images are displayed at their original size. You can set the display width (or height) of an image using the `width` (or `height`) attribute. Let's set the width of the previous image to 200 pixels:

```

```

We'll cover measurement units (such as px) in the CSS chapter.

3.4 <div> element

The <div> element is a general purpose element that is often used for grouping and organising content. It's also a good fallback element when none of the other elements are suitable. Here's an example where <div> elements are used to structure a web page:

```
<div>
  <h1>Main title</h1>
  <h2>Secondary title</h2>
</div>
<div>
  <h3>Article title</h3>
  <div>
    <p>First paragraph</p>
    <p>Second paragraph</p>
  </div>
</div>
```

Notice how the elements have been indented. This is optional and doesn't affect how the elements appear in the browser but it's useful when reading and working with the code.

The above example looks like the following when rendered in a web browser:

Main title

Secondary title

Article title

First paragraph

Second paragraph

The general purpose <div> element rendered in a web browser

See if you can relate each HTML element to each line of output.



By default the <div> elements don't affect how the HTML is rendered. In other words, the above HTML will look the same with or without the <div> elements. However you can add CSS to style the <div> elements. For example you could add margins to each <div> to add more space between the headings.

3.5 <svg> element

The <svg> element is used when you want to include SVG in your page. SVG stands for Scalable Vector Graphics and is a language for describing shapes such as lines, circles and rectangles (see the SVG chapter). The <svg> element acts as a container for your SVG shapes. For example, to add a circle to your web page you can use:

```
<svg width="800" height="600">  
  <circle r="100"></circle>  
</svg>
```

Inside the <svg> element are two **attributes**: width and height. These are similar to the width and height attributes in elements.

It's quite important to specify the SVG element's size (using width and height attributes) because its default size is quite small and shapes are cropped to the SVG's boundary.

SVG will be covered in greater depth in the SVG chapter.

3.6 Wrapping up

There are many other HTML elements but when visualising data the ones covered in this section are the most common.

You can see a comprehensive list of HTML elements at <https://developer.mozilla.org/en-US/docs/Web/HTML/Element>⁴.



Most of the examples in this book are presented using CodePen which makes it very easy to get started. However it doesn't reveal the whole story. Web pages are typically made up of individual HTML, CSS and JavaScript documents. HTML documents typically contain extra information that CodePen doesn't show. We'll look at this in more detail in the 'Tools and set-up' chapter.

3.7 Exercises

Here's some HTML that includes most of the elements covered in this chapter:

```
<div>
  <h1>Main header</h1>
  <h2>Sub header</h2>
</div>
<div>
  <h4>Article title</h4>
  <div>
    <p>First paragraph</p>
    
    <p>Second paragraph</p>
    <ul>
      <li>List item one</li>
      <li>List item two</li>
    </ul>
    <p>Third paragraph</p>
  </div>
</div>
```

You can open this example on CodePen at <https://codepen.io/createwithdata/pen/ExQGeeY>⁵.

Try the following:

1. Change the text inside the headings.
2. Change the `<h4>` & `</h4>` tags to `<h4>` and `</h4>`.
3. Add a new paragraph containing some text.
4. Change the image URL to `https://picsum.photos/id/104/384/216`
5. Change the image's width to 150 pixels.
6. Add another list item.

Notes

⁴ <https://developer.mozilla.org/en-US/docs/Web/HTML/Element>

⁵ <https://codepen.io/createwithdata/pen/ExQGeeY>

4. SVG

SVG (Scalable Vector Graphics) is a language for describing shapes. It has a similar structure to HTML but it describes shapes rather than text. The most common SVG elements when visualising data are:

Element	Description
<code><line></code>	Line
<code><rect></code>	Rectangle
<code><circle></code>	Circle
<code><text></code>	Text
<code><g></code>	Group (for grouping elements)
<code><path></code>	Path (a general purpose, free-form shape)

Each element requires a number of attributes to define its position and size. We saw in the HTML chapter that an attribute looks like `name="value"` and is inserted inside an opening tag. For example:

```
<svg width="800" height="600">
  <circle r="100"></circle>
</svg>
```

In the above snippet the `<svg>` element has two attributes `width` and `height` and the `<circle>` element has an attribute `r` with value `100`. We'll see later that the `r` attribute specifies the circle radius.

4.1 Adding SVG to a web page

SVG shape elements may be added to an HTML document by enclosing them within an `<svg>` element. For example:

```
<h3>Here's some shapes:</h3>

<svg width="800" height="600">
  <circle r="100"></circle>
  <line x1="10" y1="10" x2="100" y2="10"></line>
</svg>

<p>Above is a circle and a line.</p>
```

4.2 SVG elements

4.2.1 Line

`<line>` elements represent straight lines. They have 4 attributes:

- `x1` and `y1` which define the start of the line
- `x2` and `y2` which define the end of the line

For example:

```
<svg width="800" height="100">
  <line x1="50" y1="50" x2="200" y2="50"></line>
</svg>
```

defines a line whose start coordinate is (50,50) and end coordinate is (200,50). Each attribute is measured in pixels. The above example looks like the following:



A `<line>` element (with the `<svg>` element represented by grey outline)



In SVG the coordinate (0,0) is at the top-left of the SVG element. The x coordinate goes from left to right and the y coordinate goes from top to bottom.

Although it seems unnecessary SVG elements **do** require a closing tag!

By default, SVG lines are white, so without additional CSS rules they won't show up! You'll learn about styling using CSS in the next chapter and the Codepen example also contains a CSS rule for setting the line colour.

4.2.2 Rectangle

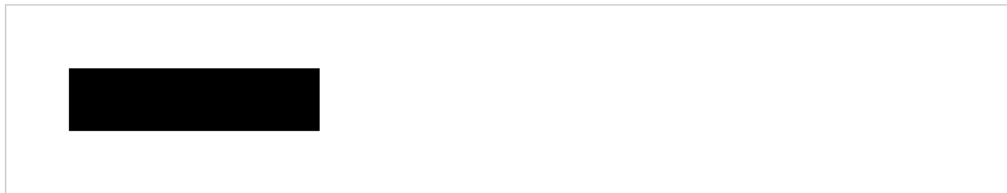
`<rect>` elements define rectangles and have 4 attributes:

- `x` and `y` which define the top left position
- `width` and `height`

For example:

```
<svg width="800" height="150">  
  <rect x="50" y="50" width="200" height="50"></rect>  
</svg>
```

defines a rectangle whose top-left corner is at (50,50) and has a width of 200 pixels and height of 50 pixels. This example looks like:



A `<rect>` element

4.2.3 Circle

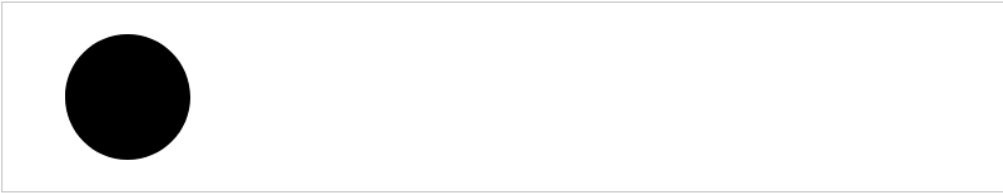
`<circle>` elements have 3 attributes:

- `cx` and `cy` which define the circle center
- `r` which defines the radius

For example:

```
<svg width="800" height="150">  
  <circle cx="100" cy="75" r="50"></circle>  
</svg>
```

This looks like:



A `<circle>` element centered at (100,75) with radius 50



Make sure you use `cx` and `cy` (and not `x` and `y`) when defining a circle's center.

4.2.4 Text

`<text>` elements have 2 attributes `x` and `y` which define the bottom-left corner of the text. Unlike most SVG elements, `<text>` elements have content between the opening and closing tags:

```
<svg width="800" height="100">  
  <text x="50" y="50">Some SVG text</text>  
</svg>
```

This example renders as:



A `<text>` element whose bottom-left is at (100,50)

4.2.5 Group

`<g>` elements are used to group SVG elements and are particularly useful when an object is made up of a few elements. They're a bit like HTML's `<div>` elements. For example a circle with text label could be represented as:

```
<svg width="800" height="600">
  <g>
    <circle cx="50" cy="50" r="20"></circle>
    <text x="75" y="55">My label</text>
  </g>
</svg>
```

This example renders as:



A `<g>` element containing two elements



In practice you should position the circle at (0,0) and transform the group as a whole (see the next section on transformations).

4.2.6 Path

`<path>` elements have a single attribute `d` which contains a list of drawing commands:

```
<svg width="800" height="100">
  <path d="M20,20 1200,0 10,50 z"></path>
</svg>
```

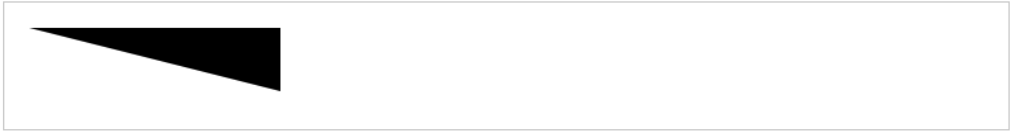
In this example, three commands are used:

- `M` which means move to a given coordinate
- `l` which means draw a line relative to the current position
- `z` which means close the shape

Therefore:

- `M20,20` means move to the coordinates `20,20`
- `1200,0` means draw a line 200 pixels to the right
- `10,50` means draw a line 50 pixels down
- `z` means close the shape

This renders as:



A `<path>` element

If you'd like more information on the path commands visit <https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Paths>⁶.

4.2.7 Exercises

Here's some SVG that includes most of the elements covered in this chapter:

```
<svg width="800" height="600">
  <line x1="20" y1="50" x2="220" y2="50"></line>
  <rect x="20" y="100" width="200" height="50"></rect>
  <circle cx="300" cy="50" r="50"></circle>
  <text x="300" y="150">SVG Text</text>
  <path d="M20,180 1200,0 10,50 z"></path>
</svg>
```

You can open this example on CodePen at <https://codepen.io/createwithdata/pen/poJqELb>⁷.



SVG Text

SVG shapes

Try the following:

1. Make the line shorter.
2. Give the circle a smaller radius.
3. Move the rectangle up or down.
4. Change the content of the `<text>` element.

SVG is a cornerstone of web-based data visualisation. Most data visualisation you see on the web (particularly if it's interactive) is SVG based. Typically they'll be made up of the elements you've covered in this section. Next time you see a data visualisation on the web see if you can identify some of the shapes that might be in use.

4.3 SVG Transforms

SVG transforms change the position, size and shape of an SVG element in a number of ways including **translation**, **scaling** and **rotation**. A translation transformation moves an element by a given x and y value. A scaling transformation makes an element larger or smaller. A rotation transformation rotates an element.

A transform can be added to a SVG element using the `transform` attribute. For example:

```
<circle transform="translate(100,100)"></circle>
```

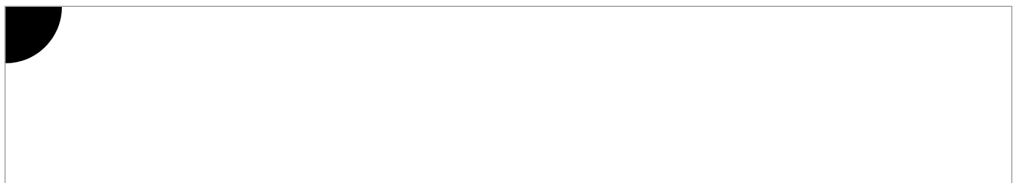
SVG transforms include `translate`, `rotate`, `scale`, `skewX` and `skewY`. In this section we'll cover `translate` and `rotate` as these are common in data visualisation.

4.3.1 Translate

Suppose you have a circle centered at $(0,0)$ with radius 50:

```
<circle r="50"></circle>
```

The default position of a circle is $(0,0)$ which corresponds to the origin at the top left of the SVG container (represented by the grey box below). This looks like:



SVG circle in default position

You can apply a translate transform to the circle using `translate(x,y)` where x and y are the amounts to move by in the x and y directions:

```
<circle r="50" transform="translate(200,75)"></circle>
```

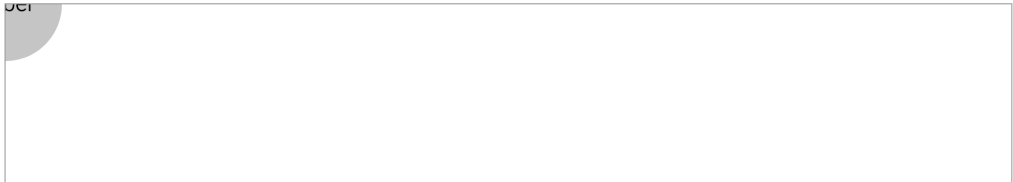
The above transform moves the circle by 200 in the x direction and 75 in the y direction. This results in a circle centered at (200,75):



SVG circle translated by (200,75)

You can apply transforms to any type of SVG element including g elements. Suppose you have a <g> element containing a circle and a text label:

```
<g>
  <circle r="50"></circle>
  <text y="6">Label</text>
</g>
```

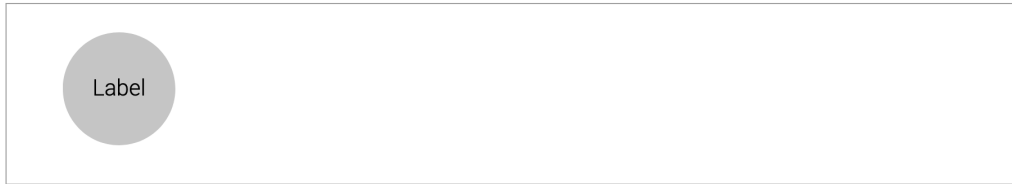


SVG circle and label before transforming

You can apply a translate transform to the group:

```
<g transform="translate(100,75)">
  <circle r="50"></circle>
  <text y="6">Label</text>
</g>
```

This has the effect of moving the circle and text element by the same amount:



SVG circle and label after translation



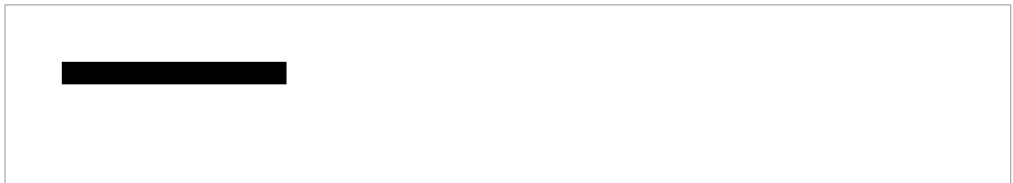
This is a very common pattern when working with SVG. You're often grouping items together and I recommend transforming the group as a whole (rather than transforming or positioning each individual element).

Note that in this example I've added some CSS to horizontally center the label. See the CSS chapter to learn more about CSS.

4.3.2 Rotate

Suppose you have a rectangle:

```
<rect x="50" y="50" width="200" height="20"></rect>
```

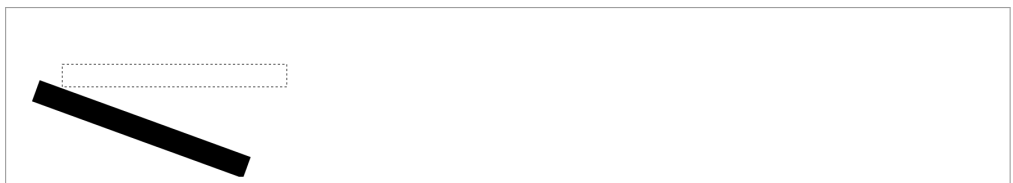


SVG rectangle without rotation

You can apply a rotate transform using `rotate(a)` which rotates an element by `a` degrees in the clockwise direction about the origin (0,0):

```
<rect x="50" y="50" width="200" height="20" transform="rotate(20)"></rect>
```

This results in:



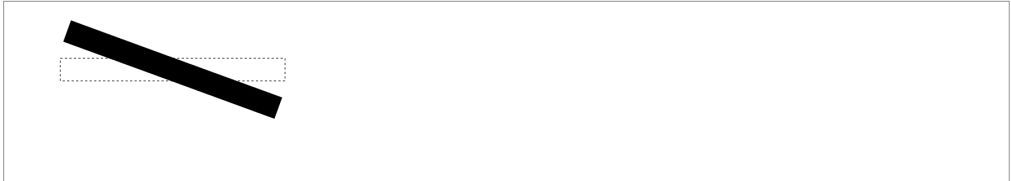
SVG rectangle with rotation (dotted rectangle represents untransformed `<rect>`)

The rectangle has rotated 20 degrees about the origin (0,0). (The origin is the top left corner of the SVG container.)

You can specify a new center of rotation by adding coordinates after the angle. The following rotates the rectangle about its center (150,60):

```
<rect x="50" y="50" width="200" height="20" transform="rotate(20 150 60)"></rect>
```

This results in:



SVG rectangle with rotation about the rectangle's center

4.3.3 Combining transforms

Transforms can be combined by writing them one after the other:

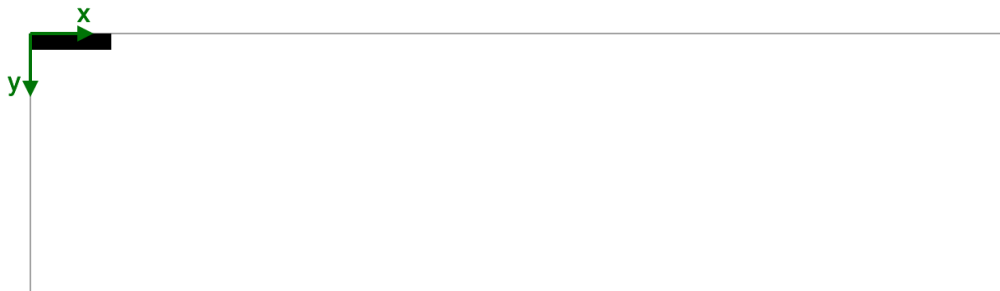
```
<rect transform="rotate(45) translate(100,0)"></rect>
```

It's important to understand that the transforms specify **transformations of the coordinate system**. For example, in the above code snippet, the coordinate system is rotated by 45 degrees then translated by (100,0).

Let's have a closer look how transforms combine. Here's a rectangle without a transform:

```
<rect width="50" height="10"></rect>
```

This looks like:

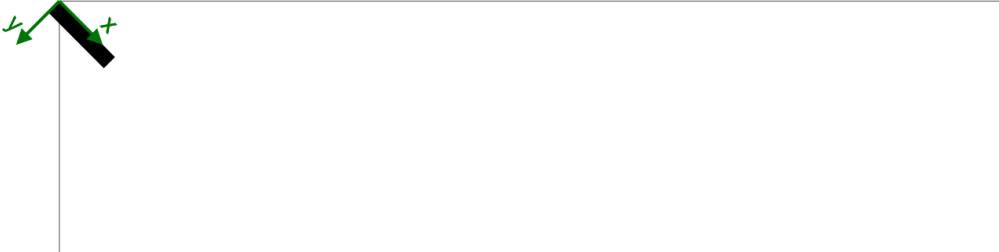


Untransformed rectangle (with coordinate system)

Now let's add a rotate transform:

```
<rect width="50" height="10" transform="rotate(45)"></rect>
```

This results in:



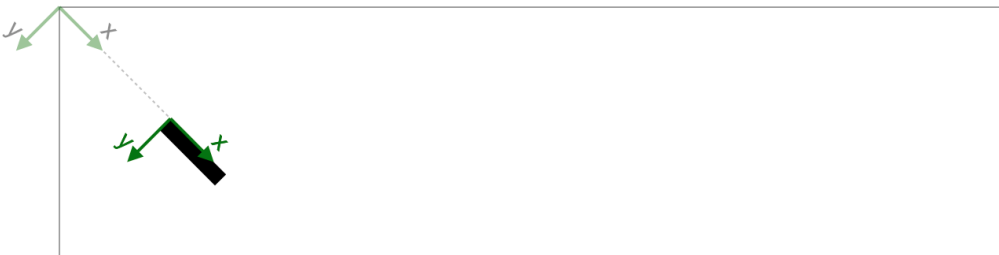
Rotated coordinate system

The coordinate system has rotated by 45 degrees and the rectangle is drawn in the **transformed coordinate system**.

Now let's add a translation of 100 pixels in the x direction:

```
<rect width="50" height="10" transform="rotate(45) translate(100,0)"></rect>
```

This looks like:



Rotated and translated coordinate system

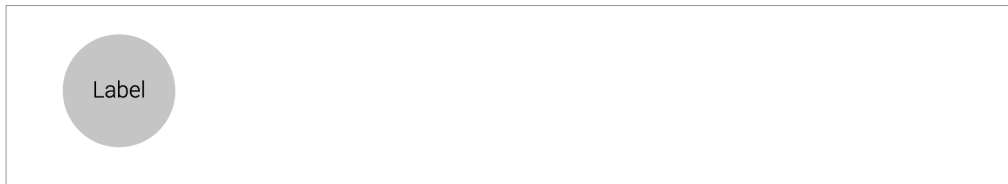
The coordinate system is 1) rotated about the origin by 45 degrees then 2) transformed by 100 pixels in the direction of the rotated x-axis. (The coordinate system with just the rotate transform is shown faintly.)

4.4 Exercises

Here's the labelled circle SVG:

```
<svg width="800" height="600">
  <g transform="translate(100,75)">
    <circle r="50"></circle>
    <text y="6">Label</text>
  </g>
</svg>
```

Navigate to <https://codepen.io/createwithdata/pen/MWwZjPp>⁸ to view this example in CodePen.



SVG circle and label after translation

Try the following (see if you can guess the effect before changing the code):

1. Change the translate transform and observe the effect on the labelled circle group.
2. Add a rotate transform after the translate and see the effect on the group.

Notes

⁶ <https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Paths>

⁷ <https://codepen.io/createwithdata/pen/poJqELb>

⁸ <https://codepen.io/createwithdata/pen/MWwZjPp>

5. CSS

CSS (Cascading Style Sheets) is a language for **styling** and **positioning** HTML and SVG elements. Styling means changing the appearance (such as **size**, **colour** and **font**) of HTML and SVG elements.

In the next section we look at the structure of CSS. We then look at CSS measurement units followed by common CSS properties.

5.1 The Structure of CSS

A CSS stylesheet is list of **rules**. Here's a single CSS rule:

```
h1 {  
  color: green;  
  font-size: 20px;  
}
```

Each rule comprises a **selector** and a list of **declarations**. In the above example the selector is `h1` and the declarations are `color: green;` and `font-size: 20px;`.

The selector defines **which** HTML or SVG elements are to be styled. The declaration defines **how** the elements are to be styled. Each declaration consists of a **property** and a **value**. In the declaration `color: green;` the property is `color` and the value is `green`.

The rule in the above example mean that on all `h1` elements the colour is set to green and the font size to 20 pixels (in height). As an example, suppose you have an `<h1>` element:

```
<h1>The title</h1>
```

Without the CSS rule applied, this looks like:

The title

`h1` element without CSS

If you apply the CSS rule:

```
h1 {  
  color: green;  
  font-size: 20px;  
}
```

you get:

The title

<h1> element styled using CSS

5.2 CSS Units

5.2.1 Size units

Sizes may be expressed using the following units:

Unit	Description	Example
px	Pixels	12px
rem	Relative to the web page's default font size	1.5rem
%	Percentage of the value of the element's parent	50%
vw	Percentage of the viewport's width	25vw
vh	Percentage of the viewport's height	100vh

Typically the web page's default font size is 16px so 1.5rem means 24px.

These units are usually sufficient when building data visualisations.

5.2.2 Colour units

Colours may be expressed in the following manner:

Type	Description	Example
Colour name	A colour name. The full list of possible colours is here ⁹ .	red, green, steelblue

Type	Description	Example
RGB	rgb followed by red, green and blue values in brackets, expressed as numbers 0 to 255.	rgb(255, 100, 255)
Hex code	A # symbol followed by red, green & blue values in hexadecimal (00 to FF). If all three values have duplicate digits e.g. #FFDD33 you can shorten it to #FD3.	#FF595E, #FD3

In my experience the above colour units are sufficient when building data visualisations. If you wish to dive deeper take a look at MDN's guide to [HTML colour](#)¹⁰.

5.3 CSS Properties

Some CSS properties apply to HTML elements, some apply to SVG elements and others apply to both HTML and SVG elements. Let's start by looking at CSS properties for HTML elements.

5.3.1 CSS properties for HTML

Some of the most common properties for HTML elements are:

Property	Description	Example values	More info
color	Text colour	red, #FF595E, rgb(255, 100, 255)	More ¹¹
background-color	Background colour	black, #CCC, rgb(0, 0, 10)	More ¹²
border	Border (around the content of the element)	solid (solid line), 1px dashed red (red dashed line)	More ¹³
padding	Width of the space between the content and the border	10px, 2rem	More ¹⁴
margin	Width of the space around the border	10px, 2rem	More ¹⁵
opacity	Opacity (or transparency)	0 (transparent), 0.5 (semi-opaque), 1 (opaque)	More ¹⁶
font-size	Text size	12px, 1.5rem	More ¹⁷

Property	Description	Example values	More info
font-family	Font family	Helvetica, Verdana, sans-serif, serif	More¹⁸
font-weight	Font weight	lighter, normal, bold	More¹⁹
width	Width	200px, 10rem, 50%	More²⁰
height	Height	100px, 5rem, 25%	More²¹



For simple cases, set `font-family` to `serif` or `sans-serif`. You can specify font names such as `Helvetica` but some fonts might not be available in everyone's browser. You can also use web font services such as [Google Fonts²²](#) but this requires additional setting up which is beyond the scope of this book.

5.3.2 Exercises

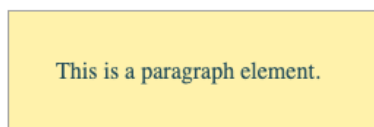
Here's some HTML:

```
<p>This is a paragraph element.</p>
```

and some CSS:

```
p {  
  font-size: 1rem;  
  color: #335C67;  
  background-color: #FFF3B0;  
  border: 1px solid #aaa;  
  padding: 2rem;  
  margin: 1rem;  
  width: 12rem;  
}
```

This looks like:



A `<p>` element styled with CSS

Navigate to <https://codepen.io/createwithdata/pen/OJQXzKo>²³ to view this example in CodePen.

Try the following:

1. Edit some of the property values (for example you could change some of the colours).
2. Add a rule for changing the opacity of `<p>` elements.
3. Increase/decrease the margin and/or padding to see how they relate.
4. Increase the border width.
5. Reduce the element width.
6. Add another `<p>` element to the HTML and observe how it looks in the result pane.

5.3.3 CSS properties for SVG

For SVG elements the most common properties are:

Property	Description	Example values	More info
<code>fill</code>	The shape's internal colour	<code>red</code> , <code>#FF595E</code> , <code>rgb(255, 100, 255)</code> , <code>none</code>	More ²⁴
<code>stroke</code>	The colour of the shape's outline	<code>red</code> , <code>#FF595E</code> , <code>rgb(255, 100, 255)</code> , <code>none</code>	More ²⁵
<code>stroke-width</code>	The width of the shape's outline	<code>1px</code> , <code>0.5px</code>	More ²⁶
<code>opacity</code>	The opacity of the shape	<code>0</code> (transparent), <code>0.5</code> (semi-opaque), <code>1</code> (opaque)	More ²⁷
<code>font-size</code>	Text size	<code>12px</code> , <code>1.5rem</code>	More ²⁸
<code>font-family</code>	Font family	<code>Helvetica</code> , <code>Verdana</code> , <code>sans-serif</code> , <code>serif</code>	More ²⁹
<code>font-weight</code>	Font weight	<code>lighter</code> , <code>normal</code> , <code>bold</code>	More ³⁰
<code>text-anchor</code>	Horizontal alignment of text element	<code>start</code> , <code>middle</code> , <code>end</code>	More ³¹

For example here's some SVG:

```
<svg width="400" height="200">
  <g transform="translate(100, 75)">
    <circle r="50"></circle>
    <text y="6">Label</text>
  </g>
</svg>
```

and some CSS:

```
circle {
  fill: #9E2A2B;
}

text {
  font-family: sans-serif;
  fill: white;
  text-anchor: middle;
}
```

This looks like:



An SVG circle and text styled with CSS

The first rule sets the colour of the circle and the second rule styles the text element. The font family, colour and horizontal alignment of the text element are set.

Navigate to <https://codepen.io/createwithdata/pen/YzeWeyY>³² to view this example in CodePen.

5.3.4 Exercise

Try the following:

1. Set the opacity of the circle to 0.5.
2. Increase the font-size of the label.

5.3.5 Overriding rules

If more than one CSS declaration applies to the same element the lower one takes precedence. For example the following two rules:

```
p {  
  font-size: 10px;  
}  
  
p {  
  font-size: 1.5rem;  
}
```

results in `p` elements having a font size of 1.5rem. (The first rule is overridden by the second.)

5.3.6 Example

Let's look at a more advanced example. Suppose you've the following HTML:

```
<h1>Main title</h1>  
<h2>Subtitle</h2>  
  
<div>  
  <p>Some description text</p>  
</div>  
  
<div>  
  <svg width="600" height="300">  
    <circle cx="150" cy="150" r="100"></circle>  
    <rect x="300" y="50" width="100" height="200"></rect>  
  </svg>  
</div>
```

and the following CSS:

```
body {  
  font-family: serif;  
}  
  
h2 {  
  color: #aaa;  
  font-family: sans-serif;  
}  
  
p {  
  color: #333;  
}
```

```
circle {  
  fill: orange;  
}  
  
rect {  
  fill: none;  
  stroke: #aaa;  
  stroke-width: 1px;  
}
```

(You can view this example on CodePen at <https://codepen.io/createwithdata/pen/LYVvBEg>³³.)

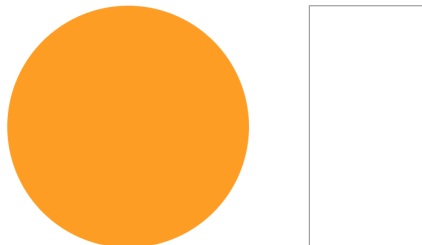
The first rule's selector is `body`. The `body` element represents all of the content so this rule sets the font family to `serif` on all elements. The next rule sets the colour and font family of `h2` elements. This `font-family` declaration overrides the `font-family` declaration of the previous rule.

Next is a rule to set the colour of `p` elements. We then set the `fill` of `circle` elements and the `fill`, `stroke` and `stroke-width` of `rect` elements. The result looks like:

Main title

Subtitle

Some description text



CSS applied to multiple HTML and SVG elements

5.4 More CSS selectors

The previous section showed how to style HTML and SVG elements using CSS. You saw how a CSS rule applies to elements of a particular type. For example:

```
p {  
  color: #333;  
}
```

sets the colour of all `<p>` elements. However what if your web page contains lots of `<p>` elements and you only want to style **some** of them?

You can do this by making CSS selectors more specific using these approaches:

- add classes and ids on HTML and SVG elements
- nested selectors
- pseudo classes

These approaches cover most cases where you need to target specific elements. Let's look at them one by one.

5.4.1 Classes and ids

There are two HTML and SVG attributes `id` and `class` that let you label particular elements in your page. This allows you to add a CSS rule that targets those elements. The `id` attribute is used where you need to target a single element and the `class` attribute is used where you need to target a number of elements.

5.4.1.1 id

The `id` attribute lets you label a single element so that you can reference it elsewhere (typically with CSS or JavaScript). For example, you might have a main menu that needs some CSS applied to it.

Let's create a simple menu and add an `id` attribute with value `main-menu`:

```
<div id="main-menu">  
  <div>Home</div>  
  <div>About</div>  
  <div>Blog</div>  
  <div>Contact</div>  
</div>
```

You can add ids to more than one element but each id value must be unique. In other words: two elements should not have the same id.

In CSS, to select an element with an id use a `#` followed by the id value:

```
#main-menu {  
  margin: 1rem;  
}
```

This adds a margin to the menu.

5.4.1.2 class

The `class` attribute is used to label **multiple** HTML and SVG elements. For example:

```
<div id="main-menu">  
  <div class="item">Home</div>  
  <div class="item">About</div>  
  <div class="item selected">Blog</div>  
  <div class="item">Contact</div>  
</div>
```

Each menu item is given a class attribute with value `item`. Class attributes can have more than one value (separated by a space). For example the third item's class attribute has two values: `item` and `selected`. This indicates that this particular `div` element is an item in a list and has been selected.

In CSS, an element with a given class value may be selected adding a **dot** followed by the **class name** to the element type:

```
div.item {  
  border: 1px solid #aaa;  
  padding: 1rem;  
}  
  
div.selected {  
  background-color: #ccc;  
}
```

This means:

- add padding and a border to each `div` element with class value `item`
- set the background colour of `div` elements with class value `selected`

You can also select all elements, regardless of type, with a given class value by omitting the element name. For example `.item` selects all elements with class value `item`.

5.4.2 Exercises

Here's a simple menu built using HTML and CSS. The HTML is:

```
<div id="main-menu">
  <div class="item">Home</div>
  <div class="item">About</div>
  <div class="item selected">Blog</div>
  <div class="item">Contact</div>
</div>
```

and the CSS is:

```
#main-menu {
  margin: 1rem;
  width: 15rem;
}

div.item {
  border: 1px solid #aaa;
  padding: 1rem;
}

div.selected {
  background-color: #ccc;
}
```

The menu looks like:

Home
About
Blog
Contact

A menu built with HTML and CSS with the second item selected.

You can view this example on CodePen at <https://codepen.io/createwithdata/pen/VwQjQpX>³⁴.

Explore the code and try the following:

1. Change the background colour of the selected item.
2. Change the font colour of the selected item.
3. Change which menu item is selected.

5.4.3 Descendant selectors

Another way to select specific elements using CSS is to use a descendant selector. This consists of two or more selectors separated by a space and it lets you select elements that are descendants of other elements. For example if you have the HTML:

```
<div id="main-menu">
  <div>Home</div>
  <div>Posts</div>
</div>
<div id="content">
  <div>Some content</div>
</div>
```

You can style the `div` elements within the `#main-menu` element using the selector `#main-menu div`. For example:

```
#main-menu div {
  padding: 1rem;
}
```

This means select all `div` elements that are descendants of the `#main-menu` element. The `div` elements outside of the menu are not selected.

Descendant selectors are a very common device and examples related to data visualisation include:

```
.bar-chart .bar {
  fill: blue;
}
.chart .legend .label {
  font-family: sans-serif;
  font-size: 0.75rem;
}
```

The first selector `.bar-chart .bar` selects elements with class `bar` that are nested within elements with class `.bar-chart`. (In other words, bar chart bars.)

The second selector `.chart .legend .label` selects elements with class `label` that are nested within elements with class `legend` that are nested within elements with class `chart`. (In other words, chart legend labels.)

Descendant selectors are handy because they let you select elements based on the nested structure of your elements. For example you can recreate the previous menu example using descendant selectors. Here's the HTML:

```
<div id="main-menu">
  <div>Home</div>
  <div>About</div>
  <div class="selected">Blog</div>
  <div>Contact</div>
</div>

<div>(This div doesn't get styled)</div>
```

And here's the CSS:

```
#main-menu {
  margin: 1rem;
  width: 15rem;
}

#main-menu div {
  border: 1px solid #aaa;
  padding: 1rem;
}

#main-menu div.selected {
  background-color: #ccc;
}
```

Notice that we're able to remove the `class="item"` attributes from each item. Instead of using `div.item` to style menu items, we use `#main-menu div`. This has simplified the HTML, making it easier to read and maintain.

Navigate to <https://codepen.io/createwithdata/pen/BaYzYZb>³⁵ to view this example in CodePen

5.4.4 Pseudo classes

Pseudo classes modify the behavior of a selector. They consist of a colon (:) followed by the pseudo class name:

```
p:hover {
  background-color: yellow;
}
```

The above changes the background colour of `<p>` elements to yellow when hovered over.

Pseudo classes commonly used when building data visualisations include:

Pseudo class	Description
:hover	Selects the element when it's being hovered over
:first-child	Selects the first sibling within a group of sibling elements. (Siblings are neighbouring elements that are nested under the same parent element.)
:last-child	Selects the last sibling within a group of sibling elements

Here's the menu example using pseudo classes to style the first item (which now represents the menu title) and the hovered item. The HTML is:

```
<div id="main-menu">
  <div>Main</div>
  <div>About</div>
  <div class="selected">Blog</div>
  <div>Contact</div>
</div>
```

and the CSS is:

```
#main-menu {
  margin: 1rem;
  width: 15rem;
}

#main-menu div {
  border: 1px solid #aaa;
  padding: 1rem;
}

#main-menu div:first-child {
  font-weight: bold;
}

#main-menu div.selected {
  background-color: #aaa;
}

#main-menu div:hover {
  background-color: #ccc;
}
```

Main
About
Blog
Contact

Styling the first item using pseudo classes

The first `div` within `#main-menu` has been made bold by using the `:first-child` pseudo class. When any item is hovered over the background colour changes because of the `:hover` pseudo-class. (In reality you wouldn't implement a menu quite like this so this example just serves as a teaching tool.)

Navigate to <https://codepen.io/createwithdata/pen/MWQeQvy>³⁶ to view this example in CodePen

5.4.5 Grouped selectors

You can apply the same set of declarations to more than one selector by separating the selectors with commas. For example, given the HTML:

```
<h1>Main heading</h1>
<h2>Sub heading</h2>
<h6>Small heading</h6>
```

The following CSS sets the font weight of all heading elements to `lighter`:

```
h1, h2, h3, h4, h5, h6 {
  font-weight: lighter;
}
```

Navigate to <https://codepen.io/createwithdata/pen/jOPoMBK>³⁷ to view this example in CodePen

As an exercise, try removing `h1` selector from the CSS rule and observing the effect.

5.5 CSS for HTML layout

CSS is also used to arrange HTML elements. For example you might want a header at the top of the page containing a title and menu. You might also want your main content to be in boxes arranged as a grid:



Dashboard layout

It takes some skill to position HTML elements using CSS. On one level it can be straightforward, but it can also be tricky. One of the challenges is that you're rarely working with a fixed size page. (It would be much easier if you were!) People might be reading your page on a mobile, tablet, laptop or large desktop monitor. You usually need to take into account all these different screen widths.

Having said this, there's a couple of recent CSS developments that make arranging your elements much easier: Flexbox and Grid. We'll give an overview of Flexbox in this book. CSS Grid is a newer development and isn't covered. It could take a whole book to cover HTML layout in detail and hopefully this section is able to give you a flavour of what's possible.

5.5.1 Block and inline elements

From a layout perspective HTML elements can be divided into two categories: **block** elements and **inline** elements.

Block elements are positioned below each other and inline elements are positioned side by side. For example if you have several `<div>` elements:

```
<div>1</div>
<div>2</div>
<div>3</div>
<div>4</div>
```

they'll appear as a vertical stack:



Block elements

If these elements are inline elements they'll be positioned side by side:



Inline elements



The size of inline elements is determined by their content. By default block elements occupy the entire width of their parent element. They can also be sized using the CSS `width` and `height` properties.

Of the elements covered in this book, heading (`<h1>` - `<h6>`), list `<ul>` and `<div>` elements are block elements (by default) and `<img>` and `<svg>` elements are inline elements (by default). You can also set whether an element is block or inline using the `display` CSS property. To set an element to block display use:

```
display: block;
```

and to set an element to inline display use:

```
display: inline;
```

Another value for `display` is `inline-block` which positions elements side by side **and** allows their width and height to be set using `width` and `height` properties. (I very rarely use `display: inline;` but occasionally use `display: inline-block;` when positioning `<div>` elements side by side.)



The direction in which block and inline elements run depends on the writing language. This book assumes the writing language is English.

5.5.2 Flexbox

The CSS Flexbox model lets you specify how items are **arranged within their parent element**. For example, let's start with this HTML:

```
<div id="container">
  <div class="item">ONE</div>
  <div class="item">TWO</div>
  <div class="item">THREE</div>
  <div class="item">FOUR</div>
</div>
```

and this CSS:

```
.item {
  padding: 1rem;
  margin: 0.25rem;
  border: 1px solid #aaa;
}
```

By default this'll render like:

ONE
TWO
THREE
FOUR

Default layout of `<div>` elements

Flexbox lets you specify properties on the container and its child elements (or 'items') to specify how the items are arranged.

To make an element a flex container set the `display` property of the element to `flex`:

```
#container {  
  display: flex;  
}
```

This results in the items being arranged in a row:



Container element (shown in grey) with flex enabled

You can now set properties on the container and/or the items to determine how the items should be arranged. These are the most common properties and values for the container:

Property	Description	Values
<code>flex-direction</code>	Determines the direction in which items will be positioned. This also determines the direction of the ‘main axis’.	<code>row</code> (items arranged in a row, main axis runs left to right), <code>column</code> (items arranged in a column , main axis runs top to bottom)
<code>flex-wrap</code>	Determines whether elements should wrap onto another row/column	<code>nowrap</code> (don’t wrap items), <code>wrap</code> (wrap items)
<code>justify-content</code>	Specifies how flex items are distributed along the main axis	<code>flex-start</code> (items placed at the start of the main axis), <code>flex-end</code> (items placed at the end of the main axis), <code>center</code> (items placed in the middle of the main axis), <code>space-around</code> (items spread along the main axis), <code>space-between</code> (items spread along the main axis with no space at the ends)

Property	Description	Values
<code>align-items</code>	Specifies how items are aligned about the main axis	<code>stretch</code> (items stretched about the main axis to fill the container), <code>center</code> (items centered about the main axis), <code>flex-start</code> (items placed at start of cross-axis), <code>flex-end</code> (items placed at end of cross-axis)

The main axis is determined by `flex-direction` and runs left to right if `flex-direction` is `row` and runs top to bottom if `flex-direction` is `column`. The cross axis is perpendicular to the main axis. See <https://css-tricks.com/snippets/css/a-guide-to-flexbox>³⁸ for more detail.

As an example, if we set `justify-content` to `space-between`:

```
#container {  
  display: flex;  
  justify-content: space-between;  
}
```

we get:



Layout with `justify-content` set to `space-between`

If we set `justify-content` to `flex-end` we get:



Layout with `justify-content` set to `flex-end`

You can also nest flex containers inside one another. For example you can build a rudimentary header bar using this approach. Here's the HTML:

```
<div id="header">
  <div id="title">My Website</div>
  <div id="menu">
    <div>Home</div>
    <div>About</div>
    <div>Contact</div>
  </div>
</div>
```

And the CSS is:

```
body {
  margin: 0;
  font-family: sans-serif;
}

#header {
  height: 4rem;
  background-color: #222;
  color: white;
  padding: 0 1rem;
  display: flex;
  justify-content: space-between;
  align-items: center;
}

#menu {
  display: flex;
}

#menu div {
  padding-left: 1rem;
}
```

The outermost `<div>` element represents the header. It has two children: a title and a menu. The header element acts as a flex container (`display` is set to `flex` on `#header`). The title and menu are arranged at opposing ends of the header by setting `justify-content` to `space-between`. The items are vertically centered within the container by setting `align-items` to `center`.

In order to arrange the menu items in a row, the menu element (`#menu`) is also made a flex container (its `display` property is set to `flex`). Thus the menu is both a flex container and a flex item.

The finished header looks like:



Website header arranged using nested flex containers

Hopefully this gives you a flavour of what Flexbox can achieve. If you'd like to learn more about Flexbox one of the best tutorials is <https://css-tricks.com/snippets/css/a-guide-to-flexbox>³⁹. I've also built a Flexbox explorer (<https://app.peterrcook.com/flexplorer/>⁴⁰) which lets you quickly explore each property interactively.

Notes

9 https://developer.mozilla.org/en-US/docs/Web/CSS/color_value

10 https://developer.mozilla.org/en-US/docs/Web/HTML/Applying_color

11 <https://developer.mozilla.org/en-US/docs/Web/CSS/color>

12 <https://developer.mozilla.org/en-US/docs/Web/CSS/background-color>

13 <https://developer.mozilla.org/en-US/docs/Web/CSS/border>

14 <https://developer.mozilla.org/en-US/docs/Web/CSS/padding>

15 <https://developer.mozilla.org/en-US/docs/Web/CSS/margin>

16 <https://developer.mozilla.org/docs/Web/CSS/opacity>

17 <https://developer.mozilla.org/en-US/docs/Web/CSS/font-size>

18 <https://developer.mozilla.org/en-US/docs/Web/CSS/font-family>

19 <https://developer.mozilla.org/en-US/docs/Web/CSS/font-weight>

20 <https://developer.mozilla.org/en-US/docs/Web/CSS/width>

21 <https://developer.mozilla.org/en-US/docs/Web/CSS/height>

22 <https://fonts.google.com/>

23 <https://codepen.io/createwithdata/pen/OJQXzKo>

- 24 https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Fills_and_Strokes
- 25 https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Fills_and_Strokes
- 26 https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Fills_and_Strokes
- 27 <https://developer.mozilla.org/docs/Web/CSS/opacity>
- 28 <https://developer.mozilla.org/en-US/docs/Web/CSS/font-size>
- 29 <https://developer.mozilla.org/en-US/docs/Web/CSS/font-family>
- 30 <https://developer.mozilla.org/en-US/docs/Web/CSS/font-weight>
- 31 <https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/text-anchor>
- 32 <https://codepen.io/createwithdata/pen/YzeWeyY>
- 33 <https://codepen.io/createwithdata/pen/LYVvBEg>
- 34 <https://codepen.io/createwithdata/pen/VwQjQpX>
- 35 <https://codepen.io/createwithdata/pen/BaYzYZb>
- 36 <https://codepen.io/createwithdata/pen/MWQeQvy>
- 37 <https://codepen.io/createwithdata/pen/jOPoMBK>
- 38 <https://css-tricks.com/snippets/css/a-guide-to-flexbox>
- 39 <https://css-tricks.com/snippets/css/a-guide-to-flexbox>
- 40 <https://app.peterrcook.com/flexplorer/>

6. JavaScript

JavaScript is the programming language of the web. It's built into the majority of web browsers and can:

- **add, modify** and **remove** HTML and SVG elements
- handle **interactivity** (for example, mouse clicks)
- **fetch** data and other resources
- perform general purpose **computation** (such as data processing)

Web-based data visualisations, particularly if they're interactive, probably use JavaScript.

JavaScript started out as a little used language for websites (it was typically used to handle a few user interactions) but is now a widespread language with hundreds of tools and libraries in its ecosystem.

Libraries such as [D3](#)⁴¹, [Leaflet](#)⁴² and [Chart.js](#)⁴³ are written in JavaScript and provide useful features for building data visualisations.

The next two sections will cover the basics of JavaScript namely **variables** and **types**. Subsequent sections will look at **operators**, **conditionals**, **iteration** and **functions**.

In order to experiment and practise JavaScript I recommend using the [jsconsole](#)⁴⁴ website. This allows you to enter JavaScript code and it evaluates it immediately. (If you're more experienced you can use your browser's [debug console](#)⁴⁵ or Node.)

When you see example code in this section you'll occasionally see `//` followed by a value:

```
10 + 5; // 15
```

The value indicates the value that the code returns. (If you type `10 + 5` into jsconsole and press return it'll show the result 15 on the next line.)

6.1 JavaScript variables

If you have a value (such as some text or a number) you can use a **variable** to point to that data. In effect you're giving the data a label so that you can refer to it elsewhere in your code. In JavaScript, a variable must be **declared** before it's used. Typically this is done using the `let` keyword:

```
let myMessage;
```



let is the modern way of declaring a variable. The older way is to use var. The differences are subtle and are beyond the scope of this book. You can read more [here](#)⁴⁶.

JavaScript statements usually end with a semicolon ;. This is optional but we'll use them in this book.

You can assign a value (such as a number or string) to a variable. Here's how to assign a string (a piece of text enclosed between quotation marks) to the variable myMessage:

```
myMessage = "Hello world!";
```

You can also assign numbers to variables:

```
let a = 10;  
let b = 20;
```

(The above code uses a shorthand where the variable is declared and assigned in a single statement.)

Any expression can be assigned to a variable.



An expression is a piece of code that evaluates to a single value such as "hello" or 2 * 10. These evaluate to "hello" and 20, respectively. An expression can also consist of variables. For example a + b evaluates to 30.

Let's assign the expression a + b to a new variable c:

```
let c = a + b;  
c; // 30
```

The a + b expression is evaluated and the result 30 is assigned to c.

6.1.0.1 Exercises

Open [jsconsole](#)⁴⁷ and type the following:

```
let a = 10;  
let b = 20;
```

Now type `a` followed by the return key. What do you see?
`jsconsole` evaluates `a` and outputs the result `10`. Now type:

```
let c = a + b;
```

Now type `c` followed by the return key. The `c` variable will be evaluated and its value `30` is output.

6.2 JavaScript data types

The most common data types in JavaScript are:

- strings (e.g. "Hello world")
- numbers (e.g. 1234, 10.567, 1.23e2)
- booleans (`true` and `false`)
- arrays (lists of values)
- objects (lists of key-value pairs)

The above are sufficient to build data visualisations. We'll look at strings, numbers and booleans in this section. There are separate chapters for arrays and objects.

6.2.1 Strings

Strings represent text. The text is enclosed by single quotes (`'`) or double quotes (`"`):

```
'A JavaScript string'  
"A string enclosed in double quotes"
```

If your text contains a single quote you use double quotes to enclose it (and vice versa):

```
"Bob's learning to code in JavaScript"
```

You can add strings together using the `+` operator:

```
let firstName = "Jolene";
let lastName = "Smith";
let fullName = firstName + " " + lastName;
fullName; // "Jolene Smith"
```

In the following examples we've a variable `a` with value "This is a string":

```
let a = "This is a string";
```

You can get the length of a string using `.length`:

```
a.length; // 16
```

You can get the `n`th character in the string using square brackets:

```
a[0]; // "T"
a[2]; // "i"
```

6.2.1.1 String manipulation

A string can be split wherever a string `s` occurs using `split(s)`. For example let's split the string by a single space character:

```
a.split(' '); // ["This", "is", "a", "string"]
```

(The square brackets represent an array which we'll cover in a subsequent section.)

You can compute the upper case version of a string using `toUpperCase()`:

```
a.toUpperCase(); // "THIS IS A STRING"
```

Note that this doesn't change (or mutate) the string. It just returns a new string and `a`'s value is still "This is a string". To make variable `a` upper case you can assign the output of `a.toUpperCase()` to `a`:

```
a = a.toUpperCase();
a; // "THIS IS A STRING"
```

There's a lower case counterpart called `toLowerCase()`.

6.2.1.2 String conversion

You can convert a string containing a number into a number using `parseFloat`. For example:

```
parseFloat('123.456'); // 123.456
parseFloat('5432');    // 5432
parseFloat('abc');      // NaN (not a number)
parseFloat('1234abc');  // 1234
parseFloat('0');        // 0
parseFloat('');         // NaN
```

Note that passing an empty string into `parseFloat` returns `NaN` which means ‘not a number’.



It’s quite common to see the `+` operator used to convert strings into numbers. For example `+'123'` evaluates to 123. You need to use with caution because converting an empty string (e.g. `+' '`) evaluates to 0. Sometimes when working with data, missing data is represented by an empty string and if this is converted using `+` it’ll evaluate as 0 which isn’t always desirable.

6.2.1.3 Exercises

Open [jsconsole](#)⁴⁸ and type the following:

```
a = "This is a string";
a.length;
a[2];
a.split(' ');
a.toUpperCase();
```

Check that your output is similar to:

```
16
"i"
["This", "is", "a", "string"]
"THIS IS A STRING"
```

6.2.2 Numbers

JavaScript has a number type for representing numbers. Integers (whole numbers) and decimals may be represented:

```
let a = 123;  
let b = 123.456;
```

The mathematical operators + (addition), - (subtraction), * (multiplication) and / division may be applied to numbers:

```
let c = 123 + 456;  
let d = 3 * 123;
```

Brackets can be used to enforce precedence:

```
(10 + 20) * 2; // 60  
10 + (20 * 2); // 50
```

The first expression evaluates as 60 and the second one 50. (An expression is a piece of code that evaluates to a single value.)

The following examples use a variable `n` with value 123.456:

```
let n = 123.456;
```

You can round numbers using `round`, `floor` and `ceil`.

`round` returns the closest integer, `floor` rounds the number down and `ceil` rounds the number up:

```
Math.round(n); // 123  
Math.floor(n); // 123  
Math.ceil(n);  // 124
```

You can get the string representation of a number using `toString()`:

```
n.toString(); // "123.456"
```

You can get the square root of a number using `Math.sqrt()`:

```
Math.sqrt(100); // 10
```

6.2.2.1 Exercises

Open [jsconsole](#)⁴⁹ and type the following:

```
let n = 123.456;
Math.round(n);
n.toString();
Math.sqrt(100);
```

Your output values should be:

```
123
"123.456"
10
```

6.2.3 Booleans

Boolean types can have two values `true` and `false`:

```
var learningCSS = true;
var learningJavaScript = true;
var learningPython = false;
```

You'll see later on (in the operators section) that booleans are returned by operators such as `===` (is equal to) and `>` (is greater than).

6.3 JavaScript arrays

Arrays represent **lists of values**. Each value can be of any type, even arrays! Arrays are one of the fundamental data structures used when working with data. (You'll see in the section on objects that it's common to represent a table of data using an array.) An array is represented by **square brackets** and each value is separated by a **comma**. Here are 4 different arrays:

```
[ 10, 20, 30 ]           // an array of numbers
[ "An", "array", "of", "strings" ] // an array of strings
[ [ 10, 20 ], [ 30, 40 ] ] // an array of arrays
[ true, true, false ]    // an array of booleans
```

6.3.1 Accessing array elements

You can access the *i*th item in an array by writing `[i]` after the array:

```
let a = [ "An", "array", "of", "strings" ];
a[0];    // "An"
a[2];    // "of"
```

Arrays are indexed from 0. So the first element in an array is referenced by `[0]`. For nested arrays such as:

```
let a = [ [ 10, 20 ], [ 30, 40 ] ];
```

you access the values by stringing together the square brackets:

```
a[0];        // [10, 20]
a[0][1];     // 20
```

`a[0]` is the first element of `a` and evaluates to `[10, 20]`.

`a[0][1]` means index 1 of `a[0]` and evaluates to 20.

6.3.2 Array operations

In the following table, these variables are used:

```
let a = [ 10, 20, 30 ];
let b = [ 40, 50, 60 ];
let c = [ "An", "array", "with", "text" ];
```

Useful array operations include the following:

Operation	Examples	Result
Get array length	<code>a.length</code>	3
Sort an array	<code>c.sort()</code>	c is now ["An", "array", "text", "with"]
Reverse an array	<code>a.reverse()</code>	a is now [30, 20, 10]
Get a portion (or 'slice') of an array. The 1st number indicates the start index of the slice. The 2nd number indicates the end index (inclusive) of the slice.	<code>c.slice(1, 2)</code>	["array", "with"]

Operation	Examples	Result
Join each element of an array with a given string	<code>a.join(' and ')</code>	"10 and 20 and 30"
Join two arrays	<code>a.concat(b)</code>	[10, 20, 30, 40, 50, 60]
Add an element to the end of an array	<code>b.push(70)</code>	b is now [40, 50, 60, 70]
Remove (and return) the last element of an array	<code>a.pop()</code>	Evaluates to 30. a is now [10, 20]

6.3.2.1 Take care when sorting

Be careful when sorting arrays of numbers. The following example:

```
let d = [11, 10, 9];
d.sort();
```

results in `d` being `[10, 11, 9]`. Which is rather surprising.

It's because JavaScript sorts as strings, even if it's an array of numbers! It's really strange. Read <https://stackoverflow.com/questions/48201502/strange-javascript-sorting-bug>⁵⁰ for more insight.

Another quirk of JavaScript is that some of the above operations change the original array while others don't. `sort` and `reverse` overwrite the array with the new order. Similarly, `push` and `pop` change the length of the original array.

The remaining operations just evaluate to a value. They don't have any effect on the array.

6.3.3 Exercises

Open [jsconsole](#)⁵¹ and type the following:

```
let a = [10, 20, 30];
a.length;
a.reverse();
a.push(0);
a;
```

The final value of `a` should be `[30, 20, 10, 0]`.

The first expression just outputs the length of `a`. The second reverses `a` so `a` is now `[30, 20, 10]`. The third statement adds `0` on to the end of `a`. The final expression `a` evaluates to `[30, 20, 10, 0]`.

6.4 JavaScript objects

JavaScript objects are **(unordered) lists of name-value pairs**. For example:

```
let carrot = {  
  foodGroup: "vegetable",  
  color: "orange"  
}
```

Each name-value pair is known as a **property**. Each property has a **name** (e.g. `foodGroup`, `color`, `id`, `height`, `data`) and a **value**. The value can be any type (including array or object). You can access properties using a dot followed by the property name:

```
carrot.foodGroup; // "vegetable"
```

An object's property may also be accessed using the property name (expressed as a string) enclosed in square brackets:

```
carrot["foodGroup"]; // "vegetable"
```

The advantage of the square bracket notation is that a variable can be used as the property name:

```
let day = "Tuesday";
```

```
let data = {  
  Monday: 20,  
  Tuesday: 30,  
  Wednesday: 10,  
  Thursday: 50,  
  Friday: 70  
};
```

```
data[day]; // 30
```

Property values can be any type. Here's an object containing arrays:

```
let data = {
  Monday: [10, 30, 20],
  Tuesday: [50, 20, 30],
  Wednesday: [40, 10, 20],
  Thursday: [30, 10, 20],
  Friday: [60, 30, 40]
};

data.Tuesday;    // [50, 20, 30]
data.Tuesday[1] // 20
```

You can also have arrays of objects. This is a **very** common structure in data visualisation:

```
let data = [
  {
    name: "Spain",
    capital: "Madrid",
    population: 46.66
  },
  {
    name: "China",
    capital: "Beijing",
    population: 1386
  }
];

data[0].name;    // "Spain"
data[1].capital; // "Beijing"
```

Given an object such as:

```
let vegetable = {
  name: "carrot",
  color: "orange"
};
```

You can set properties using the following notation:

```
vegetable.name = "broccoli";
vegetable.color = "green";
vegetable.name; // "broccoli"
```

You can use the above notation even if the property doesn't exist:

```
vegetable.family = "Brassica";
```

To get a list of an object's property names use:

```
Object.keys(vegetable); // ["name", "color", "family"]
```

This returns the array ["name", "color", "family"].

6.4.1 Exercises

Open [jsconsole](#)⁵² and create the following object:

```
let carrot = {  
  foodGroup: "vegetable",  
  color: "orange"  
}
```

Use shift + return instead of just return to insert new lines within a single statement. Or click [here](#)⁵³ for a pre-populated console.

Type the following:

```
carrot.color;
```

Check that this returns the string "orange". Now change the colour to purple by typing:

```
carrot.color = "purple";
```

Check the colour has changed by typing:

```
carrot.color;
```

This should output "purple". You can also view the object by typing:

```
carrot
```

Now add a new property named price with value 15 to the object and check that it's been added.

6.5 JavaScript operators

JavaScript operators take one or two values and compute a new value.

For example the + operator takes two numbers and computes the sum of the numbers.

In the following table these variables are defined:

```
let a = 1;
let b = 2;
```

The most common operators are:

Operator	Description	Examples	Evaluates to
+	Adds two numbers or joins two strings	1 + 2, "abc" + "def", a + b	3, "abcdef", 3
-	Subtracts two numbers	3 - 2	1
*	Multiplies two numbers	2 * 8	16
/	Divides two numbers	10 / 5	2
%	Computes the remainder of a division	11 / 5	1
!	Inverts a boolean	!true	false
>, >=, <, <=	Greater than, greater or equal to, less than and less than or equal.	5 < 6, 5 < 5, 5 <= 5, "a" > "b", "abc" < "def", a > b	true, false, true, false, true, false
===	Is equal to	10 === 10, "dogs" === "cats", a === b	true, false, false
!==	Is not equal to	10 !== 10, "dogs" !== "cats"	false, true
&&	Logical and (true if both expressions are true)	2 > 1 && 3 > 2	true
\	Logical or (true if either expression is true)	1 > 2 \ 2 > 3	false

Note that the equality operator is three equal signs ===. This operator checks that two expressions are exactly equal. There's a similar operator == which is less strict. In general I recommend using ===.

6.5.1 Ternary operator

There's an operator known as the ternary operator which deserves a special mention. The ternary operator evaluates one of two expressions depending on the value of a third expression:

```
let a = 10;
let b = 20;
let result = a > b ? "a is bigger than b" : "a is not bigger than b";
result;    // outputs "a is not bigger than b"
```

In this example the expression `a > b` is evaluated. If `true` the first expression after the question mark is evaluated. If `false` the expression after the colon `:` is evaluated.

6.5.2 Increment and decrement

The increment `++` operator takes a numeric variable and increments it:

```
let a = 10;
a++;
a;    // 11
```

It's basically shorthand for:

```
a = a + 1;
```

There's also a decrement operator `--` that decrements a variable.

6.5.3 Exercises

Open [jsconsole](#)⁵⁴ and define two variables:

```
let a = 10;
let b = 20;
```

Write expressions to:

1. Add `a` and `b`.
2. Multiply `a` and `b`.
3. Check whether `a` is greater than `b`.
4. Check whether `a` is equal to `b`.
5. Check whether `a` is equal to `10`.

6.5.3.1 Answers

Your expressions should look something like:

```
a + b;    // 30
a * b;    // 200
a > b;    // false
a === b;  // false
a === 10; // true
```

6.6 JavaScript conditionals

Conditionals are used to execute different branches of code depending on a given value. This section covers two types of conditional `if` and `switch`.

6.6.1 if

The most well known conditional is `if`. It looks like:

```
let answer;

if (a > b) {
  answer = "a is greater than b";
}
```

The `if` keyword is followed by a pair of brackets containing an expression. The expression is evaluated and if it's `true` the code in the following block is executed. If it's `false` nothing happens. (A block of code can contain multiple statements and is delimited by curly brackets.)

You can add an `else` block that'll get executed if the `if` expression is `false`:

```
let answer;

if (a > b) {
  answer = "a is greater than b";
} else {
  answer = "a is not greater than b";
}
```

You can also string a series of `else` and `if` statements together:

```
let food = {
  name: "banana",
  type: "fruit"
};

let answer;

if (food.type === "fruit") {
  answer = "It's a fruit!";
} else if (food.type === "vegetable") {
  answer = "It's a vegetable!";
} else {
  answer = "Unknown food";
}
```

6.6.2 switch

The switch statement executes different blocks of code depending on the value of an expression:

```
let food = {
  name: "banana",
  type: "fruit"
};

let answer;

switch(food.type) {
  case "fruit":
    answer = "It's a fruit!";
    break;
  case "vegetable":
    answer = "It's a vegetable!";
    break;
  default:
    answer = "Unknown food";
    break;
}
```

The switch statement is followed by an expression wrapped in brackets (in the example the expression is `food.type`). There are a number of case statements followed by possible values of the expression (such as `"fruit"` and `"vegetable"`). If there's a match, the code following the case statement is executed. The code following a case statement is executed up until the `break` statement.



It's important to include the `break` statements otherwise the following case statements will also execute. In general, add a `break` statement at the end of each case block.

You can also use multiple case statements together:

```
var food = {  
  name: "banana",  
  type: "fruit"  
};  
  
switch(food.type) {  
  case "fruit":  
  case "vegetable":  
    alert("It's a fruit or a vegetable!");  
    break;  
  default:  
    alert("Unknown food");  
    break;  
}
```



In general use `if` if the test expression returns true or false and `switch` if the test expression returns multiple values.

6.7 JavaScript functions

Functions let you wrap up code that can be called multiple times from anywhere in your code. For example suppose you need to double some numbers. You could do it this way:

```
let a = 10;  
let b = 20;  
let doubleA = a * 2;  
let doubleB = b * 2;
```

If the operation is more complicated than doubling a number this approach becomes harder to manage. An alternative approach is to define a function that takes a number and returns the double of the number:

```
function double(x) {  
  return 2 * x;  
}
```

Now we can do:

```
let a = 10;  
let b = 20;  
let doubleA = double(a);  
let doubleB = double(b);
```

There's a number of advantages of this approach:

- the intent of the code is clearer (i.e. it's more expressive)
- if the repeated operation changed, you'd only need to update one piece of code
- the function can be reused elsewhere

6.7.1 Defining functions

Functions allow a piece of code to be reused. They are defined using the function keyword:

```
function sayHello() {  
  console.log('Hello'); // console.log lets you print a value to the console  
}
```

A function is (usually) given a name. In the above example the function name is `sayHello`. You place a pair of brackets after the function name. The function's body (the code that gets executed when the function is called) is placed between curly braces. To call a function you write the name of the function followed by brackets:

```
sayHello();
```



`console.log` is a function built into JavaScript that lets you output a value. Where the value is displayed depends on where you're running JavaScript. If you're using `jsconsole` it'll appear below your code.

6.7.2 Return value

Functions can return a value. Here's a function that returns a string:

```
function getName() {  
  return "Paris";  
}
```

If you call `getName` it runs the function's code and returns the expression after the `return` keyword:

```
getName(); // "Paris"
```

The return value of a function is an expression so you can assign it to a variable:

```
var name = getName();  
name; // "Paris"
```

and use it in an `if` statement:

```
let message;  
  
if(getName() === "Paris") {  
  message = "You're in France!";  
}  
  
message; // "You're in France!"
```

6.7.3 Function arguments and parameters

You can define a list of values that are passed into your function. For example let's define a function that joins a first name and last name (with a space between):

```
function joinNames(firstName, lastName) {  
  var joinedName = firstName + " " + lastName;  
  
  return joinedName;  
}
```

This function has a list of **parameters** (`firstName` and `lastName`) inside its round brackets. These behave as variables within the function body. When the function is called a list of **arguments** is placed inside the round brackets:

```
var fullName = joinNames("Ana", "Matronic");  
fullName; // "Ana Matronic"
```

When the `joinNames` function executes each argument is assigned to the function parameters. In the above example, when the function executes `firstName` is Ana and `lastName` is Matronic.

6.7.4 Exercise

Create a function called `add10` that adds 10 to a number. Test your function by calling it with an argument 50:

```
add10(50);
```

6.8 JavaScript iteration

Iteration is the repeated execution of a piece of code. A common use case (especially when visualising data) is to execute the same piece of code on each element of an array.

This section covers the following iteration types:

- `for` loops
- `forEach`
- `map`

These three iteration types cover most cases when visualising data.

6.8.1 `for` loops

`for` loops allow you to execute a block of code a number of times:

```
for (let i = 0; i < 3; i++) {  
  console.log(i);  
}
```

The `for` statement is given three expressions:

- the first is an initialisation expression which usually initializes a counter variable to zero.

- the second is an expression that is evaluated before each iteration. If it evaluates to `true` the loop iterates again. If `false` the loop stops.
- the third expression is evaluated after each iteration. Typically the counter is incremented.

In the above code the body of the `for` loop (`console.log(i)`) is executed while `i` is less than 3. The output will be three numbers: 0, 1 and 2.

`for` loops are useful when you know how many times you want to iterate. For example if you wanted to create an array containing 5 items you could do something like:

```
let numItems = 5;
let data = [];

for (let i = 0; i < numItems; i++) {
  data[i] = 'Item-' + i;
}
```

Once the loop has finished `data` is:

```
["Item-0", "Item-1", "Item-2", "Item-3", "Item-4"]
```

6.8.2 `forEach`

Given an array, `forEach` iterates over each element of the array. You provide `forEach` with a function that gets called for each array element. Each time the function is called the array element is passed into the function:

```
let data = [10, 50, 40, 30, 20];

data.forEach(function(d) {
  console.log(d);
});
```

In this example the function is called once for each array element. The first time it's called, the first element of the array 10 is passed into the function. The second time it's called, the second element of the array 20 is passed in, and so on. In this instance the function gets called a total of five times.

Notice that the function doesn't have a name. This is an example of an **anonymous function**. These are functions that are defined 'on the fly' and typically aren't called from anywhere else in the code. They're covered in the JavaScript functions (advanced) chapter.

You can rewrite the above using a named function:

```
let data = [10, 50, 40, 30, 20];

function output(message) {
  console.log(message);
}

data.forEach(output);
```

This is more verbose, but does allow the function to be reused elsewhere.

When iterating over an array you can also use a for loop:

```
var data = [10, 50, 40, 30, 20];

for (let i = 0; i < data.length; i++) {
  console.log(data[i]);
}
```

However I recommend using `forEach` if possible because it's more expressive. ('Expressive' means that the intent of the code is clearer to another developer.)

6.8.3 map

The `map` statement takes an array and returns a new array with the same length. Each element of the new array is computed from the corresponding element in the original array. A function determines how the new values are computed.

Here's an example where `map` is used to iterate over data and outputs a new array where each element is doubled:

```
let data = [10, 50, 40, 30, 20];

let doubled = data.map(function(d) {
  return d * 2;
});

doubled; // [20, 100, 80, 60, 40]
```

For each element in `data` the function is called. For the first element 10 is passed into the function. The function returns $10 * 2$ and this'll get assigned to the first element of the new array. The second time it's called 50 is passed in and $50 * 2$ is returned and added to the new array. And so on.

`map` is very common when working with data. For example given the data:

```
let data = [
  {
    name: "Anna",
    value: "20.1"
  },
  {
    name: "Brian",
    value: "15.3"
  }
];
```

we might want to convert the `name` property to upper case and the `value` property from a string into a number.

First write a function that takes an object (with properties `name` and `value`) and returns a new object with the two properties converted:

```
function convert(d) {
  let newObject = {
    name: d.name.toUpperCase(),
    value: parseFloat(d.value)
  };

  return newObject;
}
```

Now use `map` to create a new array:

```
let converted = data.map(convert);
```

`converted` will look like:

```
[
  {
    name: "ANNA",
    value: 20.1
  },
  {
    name: "BRIAN",
    value: 15.3
  }
]
```

The above example can be rewritten using an anonymous function:

```
let converted = data.map(function(d) {  
  let newObject = {  
    name: d.name.toUpperCase(),  
    value: parseFloat(d.value)  
  };  
  
  return newObject;  
});
```



It's a matter of design whether you use an anonymous function with `map`. The advantage of anonymous functions are they keep the `map` and function in the same place so can be easier to read. The advantage of using a named function with `map` is you can reuse that function elsewhere in your code.

6.8.4 Exercises

Open [jsconsole](#)⁵⁵ and create an array of numbers:

```
let a = [10, 20, 30, 40, 50];
```

Try the following:

1. Use `forEach` to iterate over the array, outputting the square of each number. (Use `console.log` to output a number.)
2. Use `map` to create an array containing the square of each element of `a`.

Remember to use `shift + return` to create new lines within a single statement in `jsconsole`.

6.8.4.1 Answers

Using `forEach` to output the square of each number:

```
a.forEach(function(d) {  
  console.log(d * d);  
});
```

Using `map` to create an array containing the square of each element of `a`:

```
let squared = a.map(function(d) {  
  return d * d;  
});  
  
squared;    // [100, 400, 900, 1600, 2500]
```

6.9 JavaScript functions (advanced)

6.9.1 Named functions and anonymous functions

Functions usually have a name:

```
function addTen(x) {  
  return x + 10;  
}
```

This allows them to be called anywhere in your program. However when using `forEach` or `map` (see the iteration chapter) it's often convenient to use an **anonymous function**. This is a function without a name and is defined as the first argument of `forEach` or `map`. So instead of writing:

```
let data = [10, 50, 40, 30, 20];  
  
function output(message) {  
  console.log(message);  
}  
  
data.forEach(output);
```

you can write:

```
let data = [10, 50, 40, 30, 20];  
  
data.forEach(function(message) {  
  console.log(message);  
});
```

This results in more concise code and is a **very** common pattern in JavaScript. (D3 uses this pattern very often.) I highly recommend understanding this pattern properly. Study the two code examples above and try to understand the difference and similarity between the two.

6.9.2 Pure functions

A pure function is a function that doesn't have **side effects**. A side effect is when something outside of the function changes. For example:

```
let outside = 10;

function doSomething() {
  outside = 20;
}
```

Calling `doSomething` results in the variable `outside` changing. This is considered a side effect. An example of a pure function (where there are no side effects) is:

```
let outside = 10;

function doSomething() {
  return 20;
}

outside = doSomething();
```

The function just returns a value and does not affect anything outside of the function. The advantage of pure functions is that they have more predictable behaviour. They have input values (via the argument list) and a single output. You know that if you call a pure function it's not going to cause anything to change outside of that function.

6.9.3 Functions are objects

JavaScript functions behave like objects. This means you can do anything to a function that you can do to an object. For example you can assign a function to a variable:

```
let getANumber = function() {
  return 10;
}

getANumber();
```

The above is equivalent to:

```
function getANumber() {  
  return 10;  
}
```

```
getANumber();
```

You can add functions to objects:

```
let myFunctions = {  
  add: function(a, b) {  
    return a + b;  
  },  
  subtract: function(a, b) {  
    return a - b;  
  }  
};
```

```
myFunctions.add(10, 20);      // 30  
myFunctions.subtract(20, 10); // 10
```

6.9.3.1 Passing functions into functions

You can also pass functions into functions:

```
function add(a, b) {  
  return a + b;  
}
```

```
function multiply(a, b) {  
  return a * b;  
}
```

```
function getValue(a, b, operation) {  
  return operation(a, b);  
}
```

```
getValue(10, 20, add);      // 30  
getValue(10, 20, multiply); // 200
```

Here we've defined three functions:

- add takes two numbers and returns the sum
- multiply takes two numbers and returns them multiplied
- getValue takes two numbers and a function

When `getValue` is first called, the `add` function is passed in and assigned to the `operation` parameter. Thus when `operation` is called (inside `getValue`) `add` is executed.

In the second call to `getValue`, the `multiply` function is passed in meaning that `operation`'s value is `multiply`. Thus when `operation` is called, `multiply` is executed.

Navigate to <https://codepen.io/createwithdata/pen/abvJmPY>⁵⁶ to view this example in CodePen

Take your time in understanding this example. This is fairly advanced JavaScript but functions are often passed into functions when using D3.

Notes

41 <https://d3js.org/>

42 <https://leafletjs.com/>

43 <https://www.chartjs.org/>

44 <https://jsconsole.com/>

45 <https://developers.google.com/web/tools/chrome-devtools/console#javascript>

46 <https://javascript.info/var>

47 <https://jsconsole.com/>

48 <https://jsconsole.com/>

49 <https://jsconsole.com/>

50 <https://stackoverflow.com/questions/48201502/strange-javascript-sorting-bug>

51 <https://jsconsole.com/>

52 <https://jsconsole.com/>

53 <https://jsconsole.com/?let%20carrot%20%3D%20%7B%0A%20%20foodGroup%3A%20%22vegetable%22%2C%0A%20%20color%3A%20%22orange%22%0A%7D>

54 <https://jsconsole.com/>

55 <https://jsconsole.com/>

56 <https://codepen.io/createwithdata/pen/abvJmPY>

7. Tools and Set-up

This book has used CodePen for code examples. CodePen's a reasonable tool for building small applications but larger applications are typically developed locally. This means using tools and files **on your computer** (rather than on the web). This chapter outlines a typical set up for developing web applications locally. This is similar to the set-up most web developers use.

The first section gives an overview of the necessary tools and the second section walks you through setting up these tools.

7.1 Web development tools

The tools necessary for web development are:

- a **code editor**
- a **web browser**
- a **web server** (not so important if you're not loading data)

You could also add a **terminal emulator** to this list. A terminal emulator allows you to interact with your operating system via text commands. For example Terminal (on MacOS) and the Command shell (on Windows). You can work through all the material in this book without a terminal emulator so I won't cover them in this book. If you'd like to learn more I suggest a web search for 'command line tutorial'.

7.1.1 Code editor

A code editor is a desktop application that allows you to edit HTML, CSS and JavaScript files. You can use simple applications such as Notepad (on Windows) and TextEdit (on Mac). It's important that these editors work in plain text mode so that they don't insert extra characters. However I recommend using a dedicated code editor such as [Brackets](#)⁵⁷ or [Visual Studio Code](#)⁵⁸. Both of these are free and multi-platform. Advantages of using a dedicated code editor include:

- you can treat your application as a project making it easier to open and navigate

- syntax highlighting (code is coloured according to whether its a keyword, a variable etc.)
- numerous plug-ins (including web servers!) that make your life as a coder easier



Adobe dropped support for Brackets but at the time of writing it's been resurrected by the open source community. It does seem to be in a state of flux so keep checking its status if you're interested.

7.1.2 Web browser

A modern web browser such as Chrome, Firefox, Safari or Edge is recommended for web development. Create With Data books assume that **Chrome** is being used. The majority of material will apply regardless of your browser choice but there may be differences if the debugging tools are used, as these differ from browser to browser.

7.1.3 Web server

A web server typically returns (or 'serves') HTML, CSS and JavaScript files in response to requests from a web browser. For example if you type `creativewithdata.com` into a web browser the browser connects with a web server and requests files that make up that website. The web server responds by sending the requested files back to the browser.

When you create a web site on your own computer you don't necessarily need your own web server running. (You can open an HTML file directly in most web browsers.) However if you need to make requests from within your web application (as is usually the case with web based data visualisations) you will need your own local web server. (This is due to security restrictions in web browsers.) If you're an experienced developer and are comfortable setting up a local web server, feel free to skip ahead.

The options for running a local web server (starting at the easiest) include:

- Brackets editor built-in web server
- Visual Studio Code plug-ins
- Python, Ruby, PHP or Node servers

Another option which I've not personally explored is [Servez](#)⁵⁹ which is a simple GUI based web server.



Personally I use a Node server (such as [http-server](#)⁶⁰) and I recommend this if you get serious about web development. However if you're starting out Brackets is a great choice.

7.1.3.1 Brackets web server

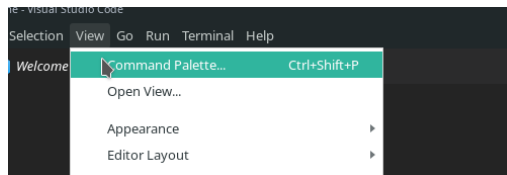
[Brackets](#)⁶¹ has a built in web server so this is by far the easiest option. This [video](#)⁶² gives a good demonstration. If you're new to code editors I recommend using Brackets.

7.1.3.2 VS Code Live Server

Visual Studio Code has a web server plug-in called [Live Server](#)⁶³.

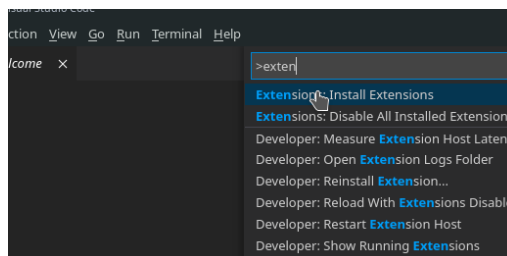
To install Live Server:

- select 'View' and then 'Command Palette...' from the menu bar



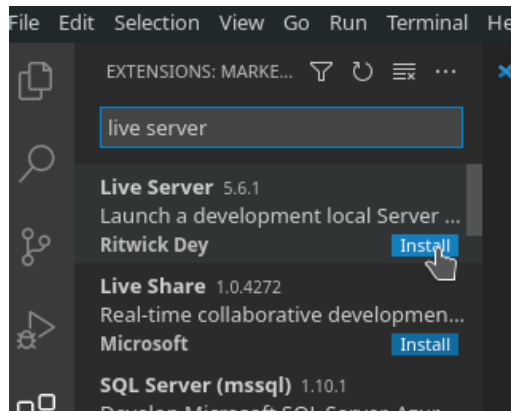
Open Command Palette in VS Code

- type 'extension' and select 'Extensions: Install Extensions'



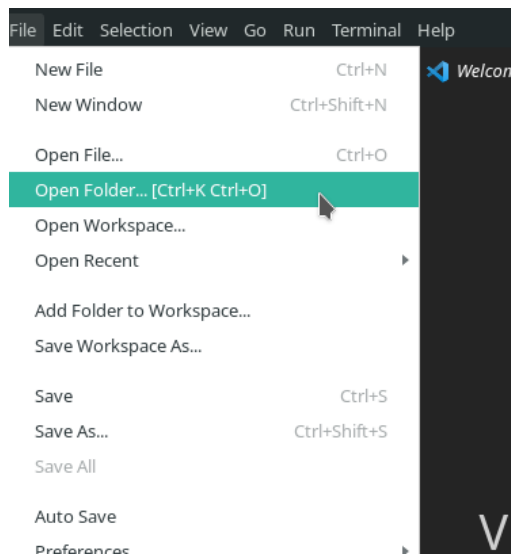
Select Extensions: Install Extensions

- search for 'live server' (by Ritwick Dey) in the extensions search box and click Install



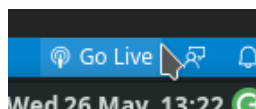
Search for 'live server'

- open a directory containing your code



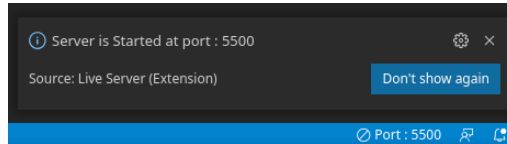
Open a directory cotaining your code

- click the Go Live button in the bottom bar



Click the Go Live button

Your web browser should then open your web application. If it doesn't, open a web browser and go to `localhost:5500`. (If a different port number to 5500 is shown in the 'Server is Started' message, use that number instead.) More detailed instructions are [here](#)⁶⁴.



Server has started

7.1.3.3 Python, Ruby, PHP, Node

If you're a Python, Ruby or PHP developer you can use the language's built in webserver. There are some useful instructions [here](#)⁶⁵. MacOS includes Python and Ruby so you can use one of these if you're a Mac user. If you're a JavaScript developer using Node you can use [http-server](#)⁶⁶.

7.2 Example set-up

This section takes you through setting up a web development environment and the creation of a simple website (using HTML, CSS and JavaScript).

The steps are:

- **create a directory** for your project
- choose and install a **code editor**
- create a simple **HTML** file
- get a local **web server** up and running
- add **CSS** and **JavaScript** files

We'll use VS Code and its Live Server extension. If you're comfortable using a different editor and web server then please feel free to do so.

7.2.1 Create a project directory

Using File Explorer (Windows) or Finder (Mac) create a directory (or folder) that'll contain your project. Name it `my_website` (or something similar). You might want to place it in another directory named `coding_projects` (or similar) if you plan on having more than one project.

7.2.2 Install a code editor

1. Download VS Code from [here](#)⁶⁷ and install
2. Run VS Code

7.2.3 Create a minimal HTML file

In this section you'll create a simple HTML file. In VS Code select **New** from the **File** menu. VS Code creates a new (unsaved and untitled) file.

The following code is just about the shortest and valid HTML document possible. It serves as a good starting point to a new web page. Copy it and paste into your new document.

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My webpage</title>
  </head>

  <body>
    <h1>Hello! </h1>
  </body>
</html>
```

Usually the only code you'll need to change is the title (between the `<title>` tags and the content (between the `<body>`) tags.



The `<!DOCTYPE>`, `lang="en"` and `charset="utf-8"` are standard bits of code that tell the browser how to interpret your HTML file. Although your webpage may appear to work without these it's best to leave them in. (See [here](#)⁶⁸ for more information about character encoding if you're interested.)

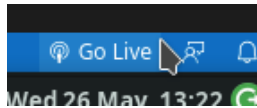
Select **Save** from the **File** menu, navigate to your project directory, name your file **index.html** and click Save. (It's important to name your file **index.html**.) Once your file is saved you should see **index.html** appear in the sidebar.

7.2.4 Install Live Server extension

Follow the instructions earlier in this chapter to install the Live Server extension.

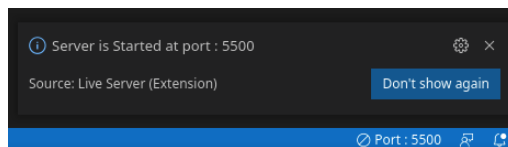
7.2.5 Start Server

If you haven't already, select **Open Folder...** from the **File** menu and select your `my_`-`website` directory. You should see your new directory appear in the sidebar. Now click on the Go Live button that appears in the bottom bar.



Click the Go Live button in VS Code

Make sure your browser is pointing at 'localhost:5500' (use whichever port number is displayed in the bottom bar).



Server has started message (with active port number displayed)

Your page should now be visible in the web browser (it should say 'Hello!'). Try editing your HTML file, saving it, then refreshing your browser. The changes you've made should be visible.

7.2.6 Add CSS and JavaScript files

In this section you'll add a CSS file and a JavaScript file to your web page. There are usually two steps to adding a file to a web page:

- create the file and save it (with extension `.css` for CSS files or `.js` for JavaScript files)
- include the file in your `index.html` file

7.2.6.1 Add a CSS file

Select **New** from the **File** menu and add the following CSS code to the new file:

style.css

```
h1 {  
  color: orange;  
}
```

Select **Save** from the **File** menu and when the dialog appears create a new directory called `css` within your project directory. Go into the `css` directory then save your new file as `style.css`. You're aiming for the following directory structure:

```
my_website  
  css  
    style.css  
  index.html
```

You now need link to your CSS file in `index.html`. Start editing your HTML file by clicking `index.html` in the left sidebar. Add the line with the `<link>` tags (highlighted below):

index.html

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="utf-8">  
    <title>My webpage</title>  
    <link rel="stylesheet" href="css/style.css">  
  </head>  
  
  <body>  
    <h1>Hello there!</h1>  
  </body>  
</html>
```

Save `index.html` and check your browser (you might need to refresh the page). The title should now be **orange**.

7.2.6.2 Add a JavaScript file

Select **New** from the **File** menu and add the following JavaScript code to the new file:

main.js

```
alert("Hello!");
```

Select **Save** from the **File** menu and when the dialog appears create a new directory within your project directory called `src`. Go into the `src` directory then save your new file as `main.js`. You're aiming for the following directory structure:

```
my_website
  css
    style.css
  src
    main.js
  index.html
```



If your files aren't in the above structure you can drag them into their correct location in the left sidebar of VS Code. You can also right click files and directories for more options.

You now need to add a link to your JavaScript file in `index.html`. Start editing your HTML file by clicking `index.html` in the left sidebar. Add the line containing the `<script>` tag (highlighted below):

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My webpage</title>
    <link rel="stylesheet" href="css/style.css">
  </head>

  <body>
    <h1>Hello there!</h1>
    <script src="src/main.js"></script>
  </body>
</html>
```

Save `index.html` and check your browser (you might need to refresh the page). An alert should appear saying 'Hello!'.

7.2.7 Summary

This section has shown how to set up a web development environment consisting of:

- a code editor
- a web browser
- a local web server

You saw how to create a simple web site consisting of an HTML file, a CSS file and a JavaScript file. This is the basis for most web applications and web-based data visualisations.

Notes

57 <http://brackets.io/>

58 <https://code.visualstudio.com/>

59 <https://greggman.github.io/servez/>

60 <https://github.com/http-party/http-server><https://github.com/http-party/http-server>

61 <http://brackets.io/>

62 <https://youtu.be/KJXdvaY9lTA?t=122>

63 <https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer>

64 <https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer>

65 <https://gist.github.com/willurd/5720255>

66 <https://www.npmjs.com/package/http-server>

67 <https://code.visualstudio.com/>

68 <http://www.w3.org/International/questions/qa-what-is-encoding>